Agoraplex Predicates Documentation

Release 0.0.3

Tripp Lilley

August 29, 2013

CONTENTS

The `predicates` module provides a variety of *predicates*, *predicate factories*, and *predicate partials*.

> "A predicate is a function that returns the truth value of some condition."
>
> —Andrew M. Kuchling, Python Functional Programming HOWTO

Predicate factories are functions which *create* new predicates based on their arguments (e.g., `_and()`, `_nargs()`). *Predicate partials* are functions created by partial application of a predicate's arguments.

CHAPTER

ONE

MOTIVATION AND APPLICATIONS

You could say that I hate Python's *lambda* syntax, and that this is just a very indirect way of expressing it. While I *do* dislike the syntax, that's really not it at all...

`predicates` is actually part of an extensible (work-in-progress) way to build matchers (or selectors, or whatever you call them). I.e., do something based on matching a value to a set of rules.

The original motivation was (is) a predicate dispatch library [1] for Python, but... something else came up.

```
>>> from predicates import (
...   _any,
...   _nis,
...   _contains,
...   isstring,
...   _and,
...   true_
... )

>>> def the_answer (*args, **kwargs):
...   return "forty-two"

>>> def the_question (*args, **kwargs):
...   return "what do you get when you multiply six by nine?"

>>> def other (*args, **kwargs):
...   return "yeah. I got nuthin'. sorry."

>>> def marvin (*args, **kwargs):
...   return "no, that's okay, i'll just sit here and rust."

>>> the_guide = (
...   (_any(_nis(exactly=42)), the_question),
...   (_any(_and(isstring, _contains("sad"))), marvin),
...   (_any(
          _and(isstring,
               _contains("life", "the universe", "everything"))),
       the_answer),
...   (true_, other)
... )

>>> def dispatch (rules, *args, **kwargs):
...   for (rule, method) in rules:
...     if rule(*args, **kwargs):
...       return method(*args, **kwargs)
```

[1] ...which explains all of the `argXXX` predicates, I hope.

```
...     raise NotImplementedError

>>> dispatch(the_guide, 42)
'what do you get when you multiply six by nine?'

>>> dispatch(the_guide, 23, 64, "how sad is he?")
"no, that's okay, i'll just sit here and rust."

>>> dispatch(the_guide,
...     "he's pondering the question.",
...     "which question?",
...     "THE question!",
...     "life. the universe. everything!",
...     "oh. that one.")
'forty-two'
```

Of course, that's a very ugly example, since it doesn't use decorators, or mine *annotations* [2]. Worse, though, it uses that brutally naïve linear probe through the predicates. While that allows giving rules precedence, it it also needlessly repeats any shared tests, and, well, it's just embarassing. So... future work on `predicates` will include a predicate compiler and an optimizer. The compiler might expand boolean predicates into Python statements, instead of requiring multiple nested function calls. The optimizer might, given a set of predicates, canonicalize them and build a tree, with each leaf being one of the original set's results.

Tomorrow Man [3] is gonna get right on that.

1.1 API Documentation

Documentation for every *predicates* API.

1.1.1 `predicates` — Predicates for functional programming

The `predicates` module provides a variety of *predicates*, *predicate factories*, and *predicate partials*.

> "A predicate is a function that returns the truth value of some condition."
>
> —Andrew M. Kuchling, Python Functional Programming HOWTO

Predicate factories are functions which *create* new predicates based on their arguments (e.g., `_and()`, `_nargs()`). *Predicate partials* are functions created by partial application of a predicate's arguments.

Naming conventions

Predicates prefixed with an underscore (_) are *predicate factories*, returning *callables*, for composition, currying, or delayed evaluation.

Predicates without a suffix, or suffixed with an underscore, are *simple predicates* (i.e., they act immediately). Most of these begin with *is* (e.g., `isstring()`, `isatom()`). If present, the underscore suffix is to avoid conflicts with keywords, in the style of `not_`).

Where names conflict with builtins or the standard library, an appropriate mnemonic prefix or suffix distinguishes them.

[2] Using anodi, for example...
[3] Not to be confused with *The* Tomorrow Man, about whom Google just told me.

There are a few places where we alias or duplicate builtins or standard library functions to present a consistently-named set of functions. E.g., `iscallable()` is equivalent to `callable()`.

Predicate composition

These functions take a sequence of predicates and return a callable which composes those predicates into a single function. These are the complement of the *Predicate application* functions.

Unless otherwise noted, the signature of the returned callable is:

fn(**args*, ***kwargs*) → bool

E.g.,

```
>>> fn = _and(isstring, isempty)
>>> fn('')
True
>>> fn("bad robot!")
False
```

We may compose the composition predicates, themselves, too. E.g.,

```
>>> fn = _and(isstring, _not(isempty))
>>> fn("bad robot!")
True
>>> fn('')
False
```

Note that *fn* is passing `*args` and `**kwargs` to its predicates all at once, instead of applying them to each argument, individually. We're effectively calling each predicate as `pred(*args, **kwargs)`, which, in this case, would be `pred('', '')`.

```
>>> fn('', '')
TypeError: _isa takes exactly 1 argument (2 given)
```

Passing two empty strings produces a `TypeError`. Since `isstring()` and `isempty()` each take only one argument, we get the `TypeError`. The *Predicate application* functions apply a predicate to each argument.

To apply multiple predicates to multiple arguments, combine the *composition* and *application* factories. E.g., to ensure that all of a function's arguments are non-empty strings:

```
>>> non_empty_string = _and(isstring, _not(isempty))
>>> fn = _all(non_empty_string)
>>> fn("bad robot!")
True
>>> fn("bad robot!", "hurley")
True
>>> fn("bad robot!", '')
False
>>> fn("bad robot!", 4)
False
```

Explanation

Conceptually, you may think of the produced callables as applying their corresponding boolean comparisons to the results of evaluating each predicate, in turn, on all of *fn*'s arguments *at once* . The example above is equivalent to:

```
>>> isstring('') and isempty('')
True
>>> isstring("bad robot!") and isempty("bad robot!")
False
```

These are also equivalent to (and, in fact, are implemented by) calling `all()`, etc., on a list comprehension which applies each of the predicates onto *fn*'s arguments. The function produced by the example above is:

```
predicates = (isstring, isempty)
def fn (*args, **kwargs):
    return all(pred(*args, **kwargs) for pred in predicates)
```

which is equivalent to:

```
def fn (*args, **kwargs):
    return all(pred(*args, **kwargs) for pred in (isstring, isempty))
```

which is equivalent to:

```
def fn (*args, **kwargs):
    predicate_results = (isstring(*args, **kwargs), isempty(*args, **kwargs))
    return all(predicate_results)
```

_and(**predicates*)
: Returns a *callable* which returns *True* if *all* `predicates` are true. This *is* short-circuiting.

_not(**predicates*)
: Returns a *callable* which returns *True* if *none* of the `predicates` are true.

_or(**predicates*)
: Returns a *callable* which returns *True* if *any* `predicates` are true. This *is* short-circuiting.

_zip(**predicates*)
: Returns a *callable* which returns *True* if each application of a predicate to its corresponding argument returns *True*. I.e., it applies `predicates[0]` to `args[0]`, `predicates[1]` to `args[1]`, and so on. Technically, this lives in the grey area between *Predicate composition* and *Predicate application*.

  It inherits the truncation behaviour of `zip()` (i.e., it truncates to the shorter of *predicates* or *args*).

  The callable ignores keyword arguments, if present.

  TODO: What's the correct mathematical term for this? The *cross-product* would be *all predicates applied to all arguments* (i.e., *n x m*). While we're on the subject, should we add a cross-product factory?

Predicate application

These functions take a single predicate and return a callable which applies that predicate to each of its arguments, applying the corresponding boolean mapping to the results. These are the complement of the *Predicate composition* functions.

Unless otherwise noted, the signature of the returned callable is:

fn(**args*, ***kwargs*) → bool

E.g.,

```
>>> fn = _all(isstring)
>>> fn()
True
>>> fn("bad robot!")
True
```

```
>>> fn("bad robot!", "jack")
True
>>> fn("bad robot!", "jack", 4, 8, 15, 16, 23, 42)
False
>>> fn(4)
False
```

_all (*predicate*)
Returns a *callable* which returns *True* if `predicate` returns *True* for *all* of its *positional* arguments.

_any (*predicate*)
Returns a *callable* which returns *True* if `predicate` returns *True* for *any* of its *positional* arguments.

_none (*predicate*)
Returns a *callable* which returns *True* if `predicate` returns *True* for *none* of its *positional* arguments.

_args (...)

_args is a special, extremely flexible, very overloaded *predicate factory* for applying predicates to a function's arguments. It is a singleton instance of `ArgSlicer`, the documentation for which covers all of the `_args()` use-cases. It is a *predicate application* because it selects a set of arguments to which to *apply* a set of predicates.

class ArgSlicer
Flexible *predicate factory* to convert `__getitem__()` slices and direct calls (i.e., `__call__()`) into *predicate partials* which apply a set of predicates to a subset of the callables' arguments.

All access is through its (unenforced) singleton instance, `_args()`, but we expose the class for subclassing, monkeypatching, etc.

Examples are the best way to explain this beast.

- `__getitem__()` access to positional arguments.

  Ensure that first positional arg (*args[0]*), if present, is a string. Imposes no constraints on other posiitional or keyword args.

  ```
  >>> fn = _args[0](isstring)
  >>> fn()
  True
  >>> fn('jack')
  True
  >>> fn(4)
  False
  >>> fn('jack', 8, kate=15)
  True
  ```

  Ensure that the first two positional args (*args[0:2]*), if present, are strings. Imposes no constraints on other positional or keyword args.

  ```
  >>> fn = _args[0:2](isstring)
  >>> fn()
  True
  >>> fn('jack')
  True
  >>> fn('jack', 8)
  False
  >>> fn('jack', 'sawyer', 15, kate=15)
  True
  ```

  Ensure that any positional args are strings. Imposes no constraints on keyword args.

```
>>> fn = _args[:](isstring)
>>> fn()
True
>>> fn('jack')
True
>>> fn('jack', 8)
False
>>> fn('jack', 'sawyer', kate=15)
```

- `__call__()` access to positional arguments:

  Ensure all positional arguments are strings. Imposes no constraints on keyword args. This is equivalent to (and, in fact, is implemented as) the `_args[:](isstring)` example, above.

```
>>> fn = _args(isstring)
>>> fn()
True
>>> fn('jack')
True
>>> fn('jack', 8)
False
>>> fn('jack', 'sawyer', kate=15)
```

- `__call__()` access to keyword arguments:

  Ensure that keyword arguments `jack` and `kate` *exist*, and that *jack* is a string, and *kate* is an integer. Imposes no constraints on positional or other keyword args.

```
>>> fn = _args(jack=isstring, kate=isint)

>>> fn()
False
>>> fn(jack='', kate=15)
True
>>> fn(jack='')
False
>>> fn(jack=4)
False
>>> fn(4, 8, jack='', kate=15)
True
```

- `__call__()` access to positional and keyword arguments:

  Ensure that keyword arguments `jack` and `kate` *exist*, that *jack* is a string, that *kate* is an integer, and that any positional arguments are strings. Imposes no constraints on other keyword args.

```
>>> fn = _args(isstring, jack=isstring, kate=isint)

>>> fn()
False
>>> fn(jack='', kate=15)
True
>>> fn("bad robot!", jack='', kate=15)
True
>>> fn("bad robot!", jack='')
False
>>> fn(jack=4)
False
>>> fn(4, 8, jack='', kate=15)
False
```

- Mixed `__getitem__()` and `__call__()` access to positional and keyword arguments:

  Ensure that keyword arguments `jack` and `kate` *exist*, that *jack* is a string, that *kate* is an integer, and that the first two positional arguments (*args[0:2]*), if present, are strings. Imposes no constraints on other positional or keyword arguments.

```
>>> fn = _args[0:2](isstring, jack=isstring, kate=isint)
>>> fn()
False
>>> fn(jack='', kate=15)
True
>>> fn("bad robot!", jack='', kate=15)
True
>>> fn("bad robot!", 'sawyer', jack='', kate=15)
True
>>> fn("bad robot!", 'sawyer', 23, jack='', kate=15)
True
>>> fn(4, "bad robot!", jack='', kate=15)
False
>>> fn("bad robot!", 'sawyer', 23, jack=4, kate='15')
False
```

Note: **TODO:** use multi-dimensional extended slice syntax to apply predicates to specific args. E.g.,

```
>>> fn = _args[0, 1, 3:5](isstring, isint, isfloat, hurley=isint)
>>> fn("bad robot!", 4, (), 8.0, 15.0, hurley=16)
True
```

Another option is to put the keyword args into the slice:

```
>>> fn = _args[0, 1, 3:5, 'hurley'](
...         isstring, isint, isfloat, isint)
>>> fn("bad robot!", 4, (), 8.0, 15.0, hurley=16)
True
```

...or is it starting to get ridiculous? I don't like the increasing distance between the arg selector (the *key* to `__getitem__()`) and the corresponding predicate. Maybe we should move to the simpler `_args(p0, p1, p2, key1=p3, key2=p4)` form? Combining `_all()` and `_apply()` would let us duplicate the behaviour of the current `_args(predicate)`, but we'd lose the ability to (easily) apply separate predicates to *ranges* of args, including *overlapping* ranges, like we get with the multidimensional slice. E.g.,

```
>>> fn = _args[0:4, 3:5](isstring, _not(isempty))
>>> fn('', '', "bad robot!")
True
>>> fn('', "bad robot!", '', 'jack', (42,))
True
>>> fn('', "bad robot!", '', 'jack', ())
False
```

Ultimately, what I'm looking for is a "single point of truth" for specifying, concisely, constraints on all of the arguments I care about.

Argument predicates

These are *predicate factories* which check constraints on the presence or absence of the arguments with which the resulting predicates are called. I.e., the new predicates evaluate the *structure* of their arguments, not their *values*.

Unless otherwise noted, the signature of the returned callable is:

fn(**args*, ***kwargs*) → bool

E.g., to test whether or not the new predicate receives at least one argument:

```
>>> fn = _npos(atleast=1)
>>> fn()
False
>>> fn("bad robot!")
True
>>> fn("bad robot!", "jack")
True
```

To test whether or not the new predicate receives at least the keyword arguments `jack` and `sawyer`:

```
>>> fn = _inkw(atleast=('jack', 'sawyer')
>>> fn(4)
False
>>> fn(jack=4, sawyer=8)
True
```

_nargs(*atleast=False*, *atmost=False*, *exactly=False*)
Returns a *callable* which returns *True* if it is called with *at least*, *at most*, or *exactly* the number of positional *and* keyword arguments specified in *atleast*, *atmost*, and *exactly*, respectively. See `_npos()` and `_nkw()` for separate constraints on positional and keyword args, respectively.

atleast and *atmost* may be combined, but *exactly* must stand alone.

_npos(*atleast=False*, *atmost=False*, *exactly=False*)
Returns a *callable* which returns *True* if it is called with *at least*, *at most*, or *exactly* the number of positional arguments specified in *atleast*, *atmost*, and *exactly*, respectively. See `_nkw()` and `_nargs()`.

atleast and *atmost* may be combined, but *exactly* must stand alone.

_nkw(*atleast=False*, *atmost=False*, *exactly=False*)
Returns a *callable* which returns *True* if it is called with *at least*, *at most*, or *exactly* the number of keyword arguments specified in *atleast*, *atmost*, and *exactly*, respectively. See `_npos()` and `_nargs()`.

atleast and *atmost* may be combined, but *exactly* must stand alone.

_inkw(*atleast=False*, *atmost=False*, *exactly=False*)
Returns a *callable* which returns *True* if it has *at least*, *at most*, or *exactly*, the keyword arguments specified. This constrains the *argument names*, while `_nkw()` constrains the *number* of arguments.

Note, too, that this does *not* constrain the keyword arguments' *values*.

atleast and *atmost* may be combined, but *exactly* must stand alone.

TODO: add a new predicate, or extend this one, to validate keyword argument values (which means we'll also need an equivalent predicate for positional args, and one for mixed positional and keyword args.)

Value predicates

These predicates test aspects of the *values* of their arguments. E.g., `isempty(val)` tests that `val` has zero length, without making demands on its type (iterability, etc.), beyond its implementing `__len__`.

isempty(*val*)
True if `val` is empty. *Empty*, here, means zero-length, not 'false-y'. I.e., `False` and `0` are *not* empty, even though they are *false* in a boolean context.

Use `truth()` and `not_()` for 'standard' truth testing.

_contains(**contents*)
Returns a *callable* which returns *True* if *each* member of `contents` is a member of its `container` argument.

The signature of the returned callable is:

fn(*container:Container*) → bool

Type predicates

These predicates test aspects of the *type* of their arguments. E.g., `isstring(val)` tests that `val` is a string (`str()` or `unicode()`), without making demands on its value (empty, non-empty, etc.)

They are *composable*, since they test only the features they need. E.g., `_and(iscallable, isiterable)` would be *True* for any class which implemented both `__call__` and `__iter__`.

Type predicate factory

_isa(*classinfo*, *docstring=None*)
A wrapper around `isinstance()` to swap the argument ordering, so it can be used as a partial.

The signature of the returned *callable* is:

fn(*obj*) → bool

If *docstring* is supplied, it will become the docstring of the new *callable*. If *docstring* is `None`, a docstring will be created based on *classinfo*.

Generated type predicates

isatom(*val*)
True if `val` looks 'atomic' (i.e., is a string, or any non-iterable). This is a naive test: any non-string iterable yields `False`.

isiterable(*obj*)
True if *obj* is *iterable*.

isnsiterable(*obj*)
True if *obj* is a non-string *iterable*

iscallable(*obj*)
True if *obj* is *callable*.

iscontainer(*obj*)
True if *obj* is a *container*.

ishashable(*obj*)
True if *obj* is *hashable*.

isiterator(*obj*)
True if *obj* is an *iterator*.

ismap(*obj*)
True if *obj* is a *mapping*.

ismmap(*obj*)
True if *obj* is a mutable *mapping*.

ismapv(*obj*)
True if *obj* is a *mapping view*.

isitemsv(*obj*)
True if *obj* is an *items view*.

iskeysv(*obj*)
True if *obj* is a *keys view*.

isvalsv(*obj*)
True if *obj* is a *values view*.

isseq(*obj*)
True if *obj* is a *sequence*.

ismseq(*obj*)
True if *obj* is a *mutable sequence*.

isset(*obj*)
True if *obj* is a *set*.

ismset(*obj*)
True if *obj* is a *mutable set*.

issized(*obj*)
True if *obj* has a `__len__()` method.

isslice(*obj*)
True if *obj* is a `slice()`.

islist(*obj*)
True if *obj* is a `list()`.

istuple(*obj*)
True if *obj* is a `tuple()`.

isstring(*obj*)
True if *obj* is a `string`.

isstr(*obj*)
True if *obj* is an `str()`.

isunicode(*obj*)
True if *obj* is a `unicode string`.

isbool(*obj*)
True if *obj* is a `bool()`.

isint(*obj*)
True if *obj* is an `int()`.

islong(*obj*)
True if *obj* is a `long()`.

isfloat(*obj*)
True if *obj* is a `float()`.

Identity predicates

These predicates test object identity (i.e., the *is* operator).

Identity predicate factory

_is (*it*, *docstring=None*)
A wrapper around `is_()` to set the docstring (which `partial()` does not).

Generated identity predicates

isnone (*obj*)
True if *obj is* `None`.

istrue (*obj*)
True if *obj is* `True`.

isfalse (*obj*)
True if *obj is* `False`.

Helpers

These functions are the foundations upon which several of the predicates are built. They may be useful when writing new predicates that are more than composition and application of the existing predicates.

_apply (*func*)
Returns a *callable* which expands its *args* and *kwargs* into the **args* and ***kwargs* of *func*. It's equivalent to `partial(apply, func)`, except that `apply()` is deprecated.

The signature of the returned callable is:

fn (*args=()*, *kwargs={}*) → object

Its principal use is to make predicates which operate on all of their arguments (i.e., **args*) operate on *any* iterable. E.g., `_apply(_all(isstring))(['jack', 'kate'])` is equivalent to `_all(isstring)('jack', 'hurley')`.

This is especially useful when testing the *contents* of arguments passed to a `_zip()` callable. E.g.,

```
>>> int_and_strings = _zip(isint, _apply(_all(isstring)))
>>> int_and_strings(42, ['jack', 'kate', 'sawyer'])
True
```

_return (*val*)
Always returns *val*.

WHY?!? Because, sometimes you need a *callable* that's just a closure over `return val`. E.g., in the 'no contents' special-case in `_contains()`.

NOTE: This is one of the few memoized factories, because we don't want a proliferation of *_return(True)* and *_return(False)* helpers (of course, that's why we have `true_()` and `false_()`, but no matter).

_nis (*atleast=False*, *atmost=False*, *exactly=False*)
Returns a *callable* which returns *True* if `n` is >= *atleast*, <= *atmost*, or == *exactly*. See `_nargs()`, `_nkw()`, etc., for example use.

The signature of the returned *callable* is:

fn (*n:number*) → bool

atleast and *atmost* may be combined, but *exactly* must stand alone.

_fnis (*func*, *atleast=False*, *atmost=False*, *exactly=False*)
Returns a *callable* which returns *True* if the result of `func(*args, **kwargs)` is >= *atleast* <= *atmost*, or == *exactly*. See `_nargs()`, `_nkw()`, etc., for example use.

atleast and *atmost* may be combined, but *exactly* must stand alone.

1.2 Indices and tables

- *genindex*
- *modindex*
- *search*

The `predicates` module provides a variety of *predicates*, *predicate factories*, and *predicate partials*.

> "A predicate is a function that returns the truth value of some condition."
>
> —Andrew M. Kuchling, Python Functional Programming HOWTO

Predicate factories are functions which *create* new predicates based on their arguments (e.g., `_and()`, `_nargs()`). *Predicate partials* are functions created by partial application of a predicate's arguments.

1.2.1 LICENSE

PYTHON MODULE INDEX

p